به نام او

دانشگاه صنعتی خواجه نصیرالدین طوسی
گروه مهندسی برق

عنوان :

تمرین مینی پروژه جهارم یادگیری ماشین

دانشجو:

آرمان مرزبان

۴۰۲۰۰۹۱۶

استاد:

دکتر علیاری

ماه و سال

تیرماه ماه ۱۴۰۳

فهرست مطالب

عنوان صفحه

فهرست شکل‌ها

بخش اول

سوال اول

۱-۱- پرسش یک: حل دنیایWumpus .

*Wumpus d*یک مسئله کلاسیک در هوش مصنوعی و یادگیری تقویتی است که شامل یک محیط مبتنی بر شبکه است که در آن یک عامل باید برای یافتن طلا حرکت کند و در عین حال از خطراتی مانند چاله ها و *Wumpus* اجتناب کند.

- اهداف پیمایش در شبکه *Grid*: عامل باید یاد بگیرد که به طور موثر در شبکه حرکت کند.
- اجتناب از خطرات: عامل باید یاد بگیرد که از چاله ها و *Wumpus* اجتناب کند.
- جمع آوری طلا: عامل باید طلا را پیدا کرده و جمع آوری کند.
- کشتن *Wumpus*: عامل می تواند برای کشتن *Wumpus* تیری شلیک کند و آن را به عنوان تهدید از بین ببرد.
- راه اندازی محیط شبکه: یک شبکه ۴*۴ که در آن هر سلول می تواند خالی باشد، حاوی یک گودال، *Wumpus*
- یا طلا باشد.
- فضای اکشن ها: حرکت به بالا، پایین، چپ، راست.
- یک فلش را در هر یک از چهار جهت (بالا، پایین، چپ، راست) شلیک کنید (امتیازی).
- تصورات: *Wumpus* در شبکه با هر تغییر اکشن به اندازه یک خانه در راستای چپ، راست، بالا یا پایین حرکت
- می کند (امتیازی).
- فضای *Reward*:

– ۱۰۰+ برای گرفتن طلا

– – ۱۰۰۰ برای افتادن در گودال یا خورده شدن توسط *Wumpus*

– ۵۰+ برای کشتن *Wumpus*) امتیازی)

– – ۱ برای هر حرکت

- تعریف محیط: یک شـــبکه ۴×۴ با موقعیت های دلخواه برای چاله ها، *Wumpus* و طلا ایجاد کنید. حالت اولیه و حالت های ممکن را بعد از هر عمل تعریف کنید.
- تنظیم پارامترها:

– نرخ یادگیری : ۰.۱

– ضریب تخفیف: ۰.۹

– نرخ اکتشاف: از ۱.۰ شروع می شود و در طول زمان کوچک میشود.

با توجه به موارد کلی گفته شده راجع به مسئله، موارد زیر را پاسخ دهید.

آ. برای این مســـئله یک بار با روش *Q learning*-و یک بار با روش*Q Deep learning*- عاملی را طراحی کرده و آموزش دهید.

پاسخ:

مرحله ۱: تعریف محیط

- ابتدا محیط ۴×۴ *Wumpus World* را ایجاد کنید. به عنوان مثال، فرض کنید محیط به این صورت است:
- وضعیت (۰,۰): عامل شروع می‌کند.
- وضعیت (۱,۳): طلا
- وضعیت (۲,۲): چاله
- وضعیت (۳,۱): *Wumpus*

در این حالت جهت تعریف محیط و وضـعیت حرکت *agent* به سـمت راسـت، بالا، پایین و چپ از قطعه

کد زیر استفاده می نماییم:

```
# Environment design
grid_size = 4
actions = ['up', 'down', 'left', 'right']

# Helper functions to define the environment
def get_next_state(state, action):
    x, y = state
    if action == 'up' and y < grid_size - 1:
        y += 1
    elif action == 'down' and y > 0:
        y -= 1
    elif action == 'left' and x > 0:
        x -= 1
    elif action == 'right' and x < grid_size - 1:
        x += 1
    return (x, y)

def get_reward(state):
    if state == (3, 3):  # Suppose gold is located at (3, 3).
        return 100
    elif state in [(1, 1), (2, 2), (3, 1)]:  # Suppose the
pits are located in these positions
        return -1000
    elif state == (1, 3):  # Suppose Wumpus is located at (1,
3).
        return -1000
    else:
        return -1
```

در گام دوم الگوریتم *Q-Learning* را پیاده سازی می نماییم و ابتدا پارامترهای الگوریتم را توسط قطعه

کد زیر تعریف می نماییم:

```
alpha = 0.1  # Learning rate
gamma = 0.9  # Discount factor
epsilon = 0.1  # Discovery rate

# Create a Q table with zeros
Q = np.zeros((grid_size, grid_size, len(actions)))
```

جدول ماتریس Q را به صورت زیر و با استفاده از کد زیر و تعداد اکشن ها تعریف می کنیم:

```
# Create a Q table with zeros
```

```
Q = np.zeros((grid_size, grid_size, len(actions)))
```

تابع پاداش را نیز با توجه به خواسته مسئله به صورت کد زیر تعریف می کنیم:

```
def get_reward(state):
    if state == (3, 3):  # Suppose gold is located at (3, 3).
        return 100
    elif state in [(1, 1), (2, 2), (3, 1)]:  # Suppose the
pits are located in these positions
        return -1000
    elif state == (1, 3):  # Suppose Wumpus is located at (1,
3).
        return -1000
    else:
        return -1
```

تابع اکشن های *agent* را نیز با استفاده از قطعه کد زیر پیاده سازی می کنیم:

```
def choose_action(state):
    if np.random.rand() < epsilon:
        return np.random.choice(actions)
    else:
        return actions[np.argmax(Q[state[0], state[1]])]
```

با استفاده از قطعه کد زیر نیز الگوریتم Q را به روز رسانی می کنیم

```
def update_q(state, action, reward, next_state):
    action_index = actions.index(action)
    best_next_action = np.argmax(Q[next_state[0],
next_state[1]])
    td_target = reward + gamma * Q[next_state[0],
next_state[1], best_next_action]
    td_error = td_target - Q[state[0], state[1], action_index]
    Q[state[0], state[1], action_index] += alpha * td_error
```

جهت آموزش مدل *Agent* نیز از قطعه کد زیر استفاده می کنیم:

```
def train_agent(episodes):
    rewards = []
    for _ in range(episodes):
        state = (0, 0)
        total_reward = 0
        while True:
            action = choose_action(state)
            next_state = get_next_state(state, action)
            reward = get_reward(next_state)
```

```
            update_q(state, action, reward, next_state)
            state = next_state
            total_reward += reward
            if reward == 100 or reward == -1000:
                break
        rewards.append(total_reward)
    return rewards
```

در بخش بعد نیز جهت پیاده سـازی و آموزش عامل با اسـتفاده از *Deep Q-Learning* گام های زیر را انجام می دهیم.

در گام اول با استفاده از کتابخانه *sklearn* شبکه عصبی یا مدل را توسط قطعه کد زیر ایجاد می کنیم

```
# Model definition
def build_model():
    model = MLPRegressor(hidden_layer_sizes=(16, 10),
activation='relu', solver='adam', learning_rate_init=0.1,
max_iter=10)
    # We train the model with the initial data
    model.fit(np.array([[0, 0]]), np.array([[0, 0, 0, 0]]))
    return model
```

با استفاده از کد زیر نیز حافظه تجربه را برای عامل ایجاد می کنیم:

```
# Definition of DQN factor
class DQNAgent:
    def __init__(self):
        self.model = build_model()
        self.memory = deque(maxlen=2000)
        self.gamma = 0.9  # Discount factor
        self.epsilon = 0.1  # Discovery rate
        self.epsilon_decay = 0.995
        self.epsilon_min = 0.01
        self.learning_rate = 0.1

    def remember(self, state, action, reward, next_state,
done):
        self.memory.append((state, action, reward, next_state,
done))

    def act(self, state):
        if np.random.rand() <= self.epsilon:
            return random.choice(actions)
        state = np.reshape(state, [1, 2])
        try:
            act_values = self.model.predict(state)
```

```
        except NotFittedError:
            act_values = np.zeros((1, len(actions)))
        return actions[np.argmax(act_values[0])]

    def replay(self, batch_size):
        minibatch = random.sample(self.memory, batch_size)
        X = []
        y = []
        for state, action, reward, next_state, done in
minibatch:
            target = reward
            if not done:
                next_state = np.reshape(next_state, [1, 2])
                target = (reward + self.gamma *
np.amax(self.model.predict(next_state)[0]))
            state = np.reshape(state, [1, 2])
            target_f = self.model.predict(state)[0]
            target_f[actions.index(action)] = target
            X.append(state[0])
            y.append(target_f)
        self.model.fit(X, y)
        if self.epsilon > self.epsilon_min:
            self.epsilon *= self.epsilon_decay

    def train(self, episodes, batch_size):
        rewards = []
        for e in range(episodes):
            state = np.reshape((0, 0), [1, 2])
            total_reward = 0
            for time in range(50):  #
                action = self.act(state)
                next_state = get_next_state(tuple(state[0]),
action)
                reward = get_reward(next_state)
                done = reward == 100 or reward == -1000
                next_state = np.reshape(next_state, [1, 2])
                self.remember(state, action, reward,
next_state, done)
                state = next_state
                total_reward += reward
                if done:
                    break
                if len(self.memory) > batch_size:
                    self.replay(batch_size)
            rewards.append(total_reward)
        return rewards
```

در گام بعد از ایجاد حافظه، با استفاده از قطعه کد زیر دو مدل Q و $DeeP\ Q$ را آموزش می دهیم.

```
# if __name__ == "__main__":
episodes = 20

    # Q-Learning
q_rewards = train_agent(episodes)

    # Deep Q-Learning
agent = DQNAgent()
dqn_rewards = agent.train(episodes, 32
```

ب. عملکرد *Policy*:

- پاداش تجمعی را در اپیزودها برای هر دو عامل Q-*learning* و DQN ترسیم کنید. چگونه عملکرد عامل در طول زمان بهبود می یابد؟

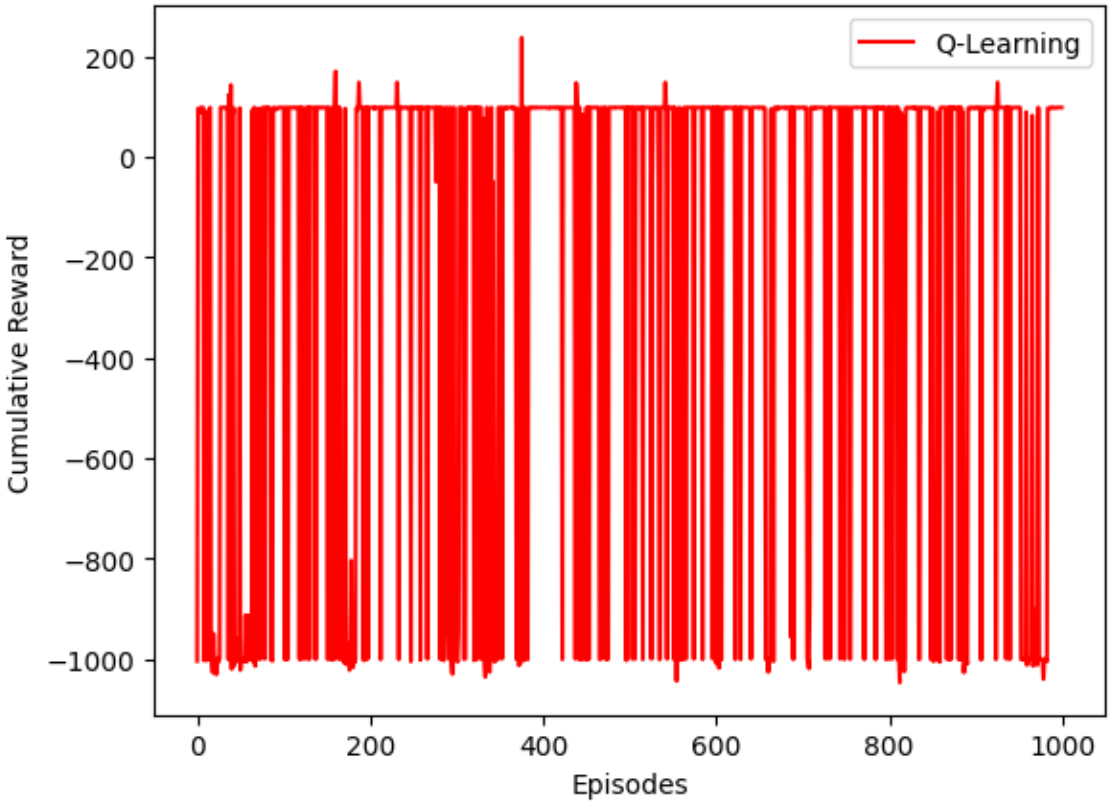
در این قسمت جهت رسم میزان پاداش ها ابتدا با استفاده از کد زیر *agent* را با دو الگوریتم Q و $DeepQ$ به ازای ۱۰۰۰ اپیزود آموزش می دهیم:

```
# if __name__ == "__main__":
episodes = 1000
Batch_size = 64
    # Q-Learning
q_rewards = train_agent(episodes)

    # Deep Q-Learning
agent = DQNAgent()
dqn_rewards = agent.train(episodes, Batch_size)
```

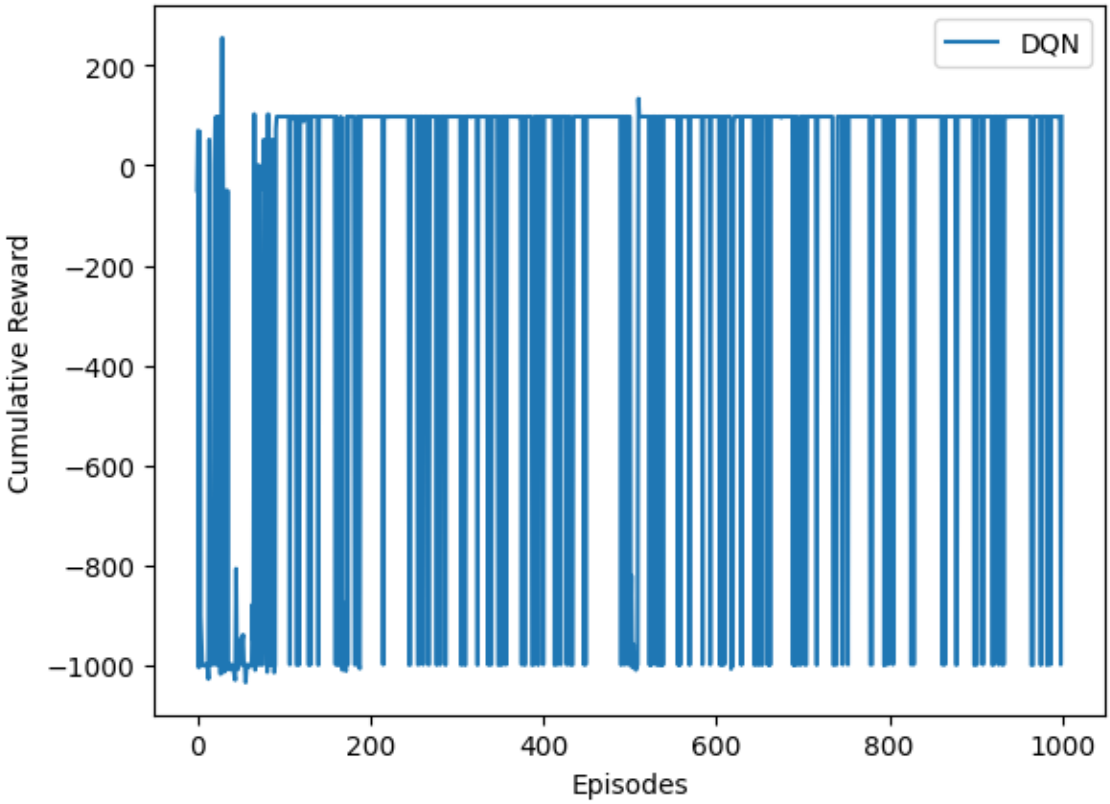
سپس جهت رسم پاداش ها در هر اپیزود برای هر دو الگوریتم از کد زیر استفاده می کنیم:

```
import matplotlib.pyplot as plt

# Draw cumulative rewards
plt.plot(range(episodes), q_rewards, label='Q-Learning')
plt.plot(range(episodes), dqn_rewards, label='DQN')
plt.xlabel('Episodes')
plt.ylabel('Cumulative Reward')
plt.title('Cumulative Reward over Episodes')
plt.legend()
plt.show()
```

که نتیجه به صورت شکل زیر بدست آمده است:



شکل ۱: پاداش عامل در هر اپیزود با استفاده از روش Q



شکل ۲: پاداش عامل در هر اپیزود با استفاده از روش DeePQ

باتوجه به شکل ۱و۲ مشاهد می شود که عامل به ازای پاداش های مثبت و منفی که دریافت کرده است هم توانسته است به طلا دست یابد(زمانی که پاداش ۱۰۰ است) هم به Wumpus شلیک کند(زمانی که پاداش ۵۰ دریافت کرده است) و هم زمانی که توسط آن خورده شده است(تنبیه -۱۰۰۰). اما چیزی که از دو شکل قابل استباط است میزان پاداش ها مدام در بین -۱۰۰۰ و ۱۰۰ در حال نوسان است و عامل با وجود اینکه مسیر را برای رسیدن به طلا پیدا کرده است، اما به دلیل آنکه Wumpus نیز حرکت می کند گاهی توسط آن خورده می شود. بنابراین برای اینکه مسیر را به خوبی یاد بگیرد بایستی تعداد اپیزودها و پارامترهای الگوریتم ها به خوبی تنظیم شوند.

- میانگین پاداش در هر اپیزود را برای هر دو عامل پس از ۱۰۰۰ اپیزود مقایسه کنید. کدام الگوریتم عملکرد بهتری داشت؟

میانگین پاداش در هر اپیزود با استفاده از هر د الگوریتم توسط قطعه کد زیر بدست می آید:

```
q_avg_reward = np.mean(q_rewards)
dqn_avg_reward = np.mean(dqn_rewards)

print(f"Average reward for Q-Learning: {q_avg_reward}")
print(f"Average reward for DQN: {dqn_avg_reward}")
```

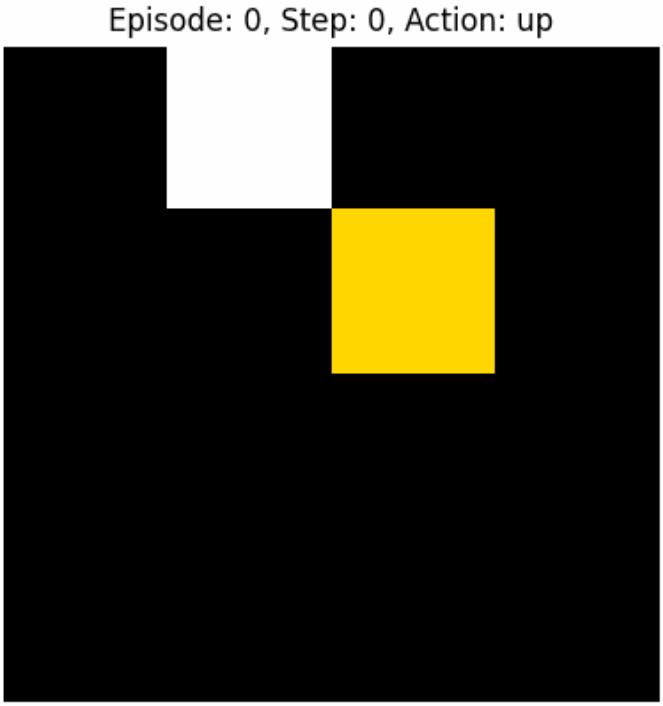
```
```

و نتیجه میانگین به صورت زیر بدست آمده است:

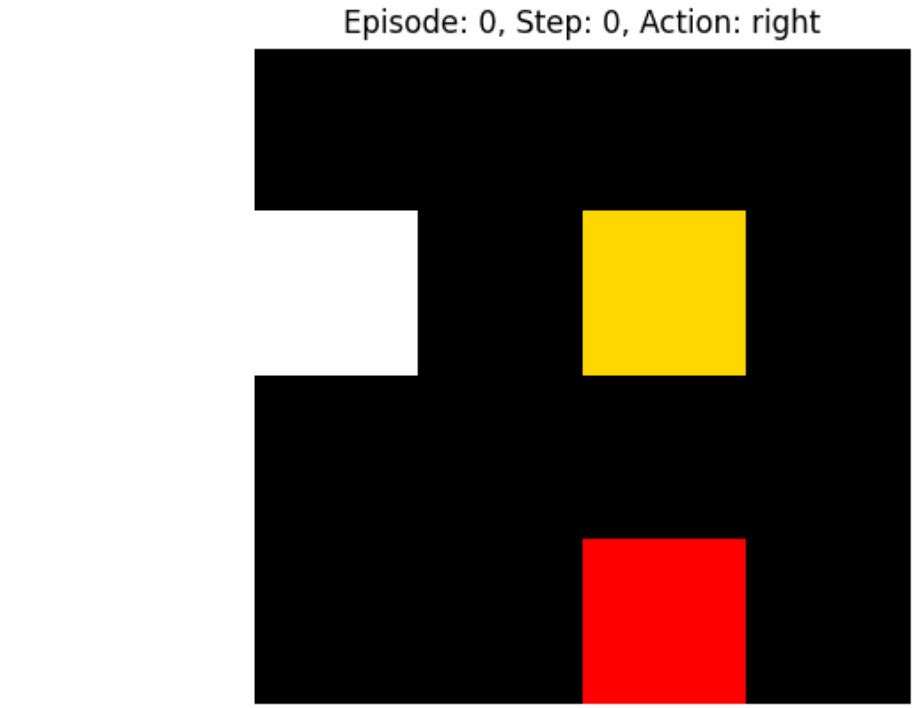
```
Average reward for Q-Learning: -163.728
Average reward for DQN: -142.73
```

ملاحظه می شود میانیگن پاداش منفی الگوریتم DeepQ بهتر از روش Q است و این الگوریتم عملکرد نسبتا بهتری نسبت به الگوریتم Q داشته است.

نحوه حرکت عاملها و پیدا کردن مسیر طلا به صورت شکل gift زیر در ۱۰ اپیزود است:

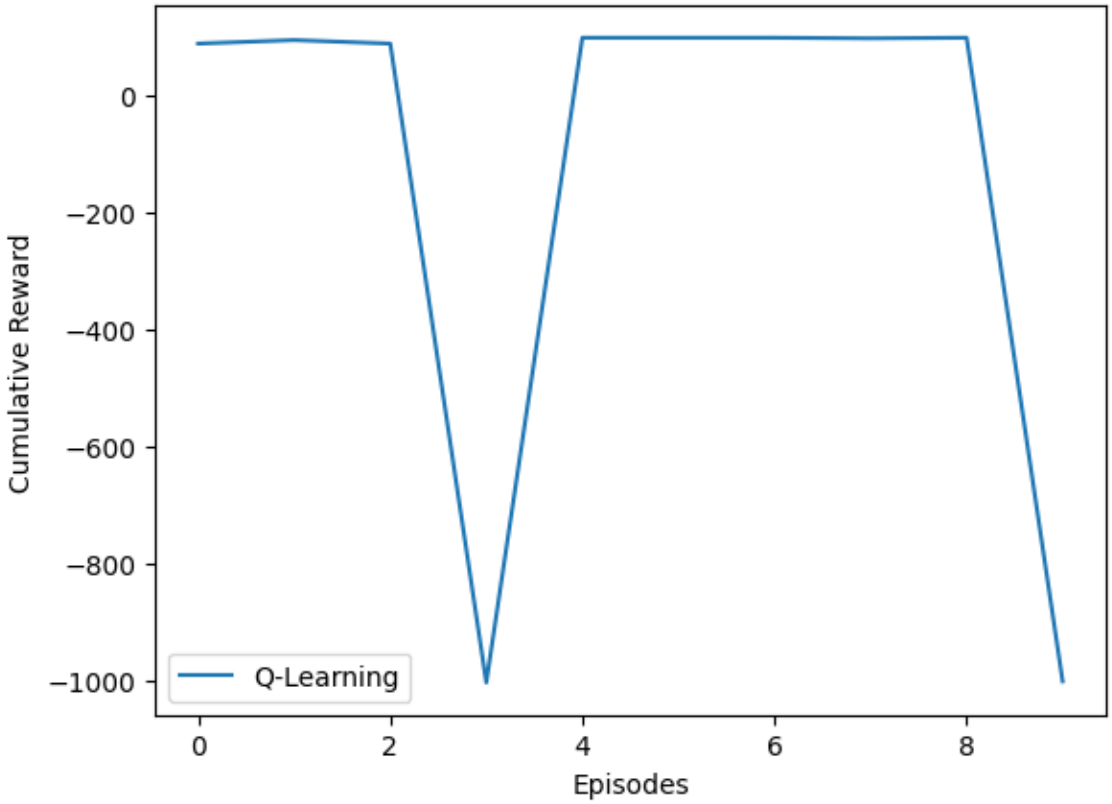


شکل ۳: نحوه حرکت عامل در ۱۰ اپیزود با الگوریتم Q(عامل رنگ سفید، طلا نارنجی و Wumpus قرمز) لینک آنلاین

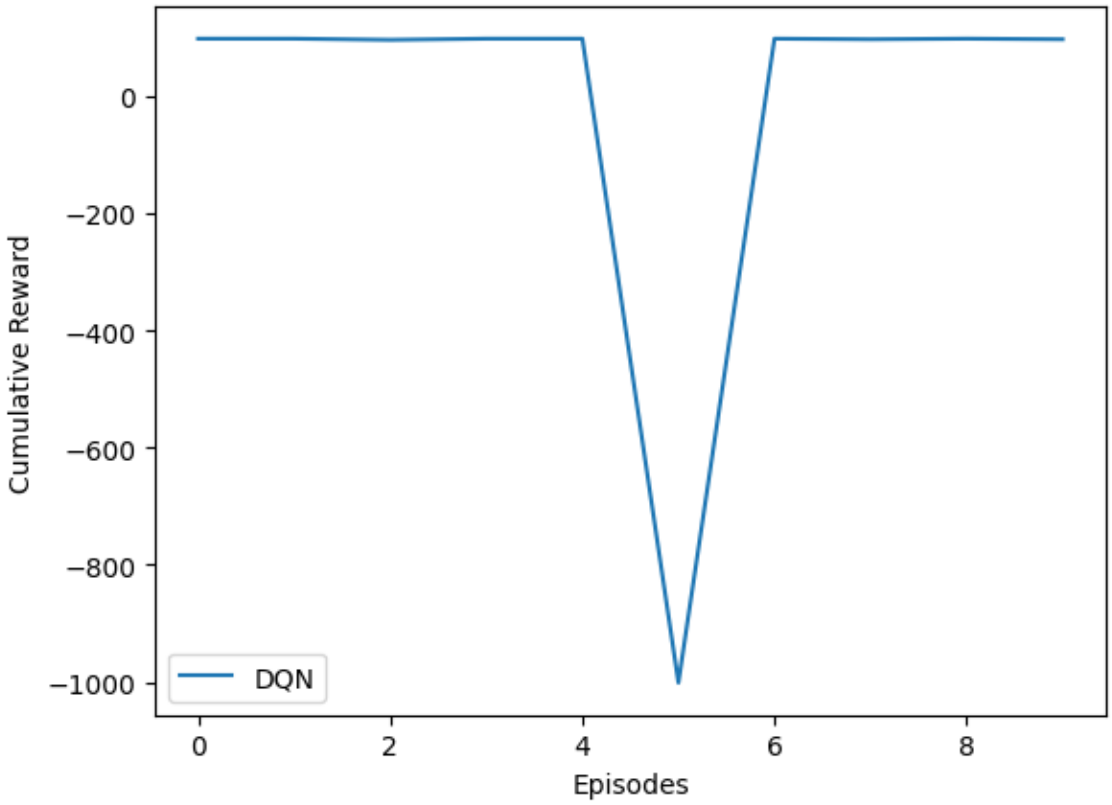


شکل ۴: نحوه حرکت عامل در ۱۰ اپیزود با الگوریتم DeePQ(عامل رنگ سفید، طلا نارنجی و Wumpus قرمز)
لینک آنلاین

نمودار پاداش ها در اینحالت به ازای ۱۰ اپیزود به صورت شکل زیر است



شکل ۵: نمودار پاداش عامل در ۱۰ اپیزود با الگوریتم **Q**



شکل ۶: نمودار پاداش عامل در ۱۰ اپیزود با الگوریتم **DeePQ**

مشاهده می شود که الگوریتم DeePQ در این حالت عملکرد بهتری داشته و به صورت پایدار تر توانسته است مسیر یافتن طلا را پیدا کند.

ج. بحث کنید که چگونه نرخ اکتشاف اپسیلون بر فرآیند یادگیری تأثیر می گذارد. وقتی اپسیلون بالا بود در مقابل وقتی کم بود چه چیزی را مشاهده کردید؟

نرخ اکتشاف ϵ یکی از پارامترهای کلیدی در الگوریتم‌های یادگیری تقویتی است که تعادل بین اکتشاف (جستجوی اکشن‌های جدید) و بهره‌برداری (استفاده از اکشن‌هایی که تاکنون بهترین نتیجه را داده‌اند) را تعیین می‌کند. در اینجا، به بررسی تأثیر نرخ اکتشاف ϵ\epsilon بر فرآیند یادگیری عامل می‌پردازیم و مقایسه‌ای بین حالت‌های مختلف این پارامتر انجام می‌دهیم.

- **ϵ بالا**

وقتی ϵ بالا تنظیم می‌شود (مثلاً ۰.۹)، عامل تمایل بیشتری به اکتشاف محیط دارد. در این حالت:

- **اکتشاف بیشتر** :عامل بیشتر به دنبال اقدامات تصادفی می‌رود تا محیط را بهتر بشناسد. این امر به خصوص در مراحل اولیه یادگیری مفید است، زیرا به عامل کمک می‌کند تا اطلاعات بیشتری درباره محیط کسب کند.
- **پاداش‌های متغیر** :به دلیل اکتشاف بیشتر، پاداش‌های عامل ممکن است به شدت متغیر باشد. ممکن است عامل به سرعت پاداش‌های بالا را پیدا کند یا برعکس، به چاله‌ها و خطرات بیشتری برخورد کند.
- **کندی در بهره‌برداری** :به دلیل اکتشاف بیشتر، عامل زمان بیشتری می‌برد تا به استراتژی بهینه برسد و از اکشن‌های بهینه بهره‌برداری کند.

- **ϵ پایین**

وقتی ϵ پایین تنظیم می‌شود (مثلاً ۰.۱)، عامل تمایل بیشتری به بهره‌برداری از اکشن‌های موجود دارد. در این حالت:

- **اکتشاف کمتر** :عامل کمتر به دنبال اقدامات تصادفی می‌رود و بیشتر از اکشن‌هایی که تاکنون بهترین نتیجه را داده‌اند، استفاده می‌کند.
- **پاداش‌های پایدارتر** :به دلیل بهره‌برداری بیشتر، پاداش‌های عامل به مرور زمان پایدارتر و کمتر متغیر می‌شود.
- **رشد سریع‌تر در بهره‌وری** :عامل سریع‌تر به استراتژی بهینه نزدیک می‌شود و شروع به کسب پاداش‌های بالاتر و پایدارتر می‌کند.

- **تغییرات ϵ در طول زمان**

به طور خلاصه، نرخ اکتشاف ϵ تأثیر مهمی بر فرآیند یادگیری عامل دارد ϵ . بالا به عامل کمک می‌کند تا محیط را بهتر بشناسد و به اکتشاف بپردازد، در حالی که ϵ پایین به عامل کمک می‌کند تا از استراتژی‌های بهینه بهره‌برداری کند و پاداش‌های پایدارتر کسب کند. کاهش تدریجی ϵ در طول زمان یکی از روش‌های متداول و مؤثر برای دستیابی به تعادل بهینه بین اکتشاف و بهره‌برداری است.

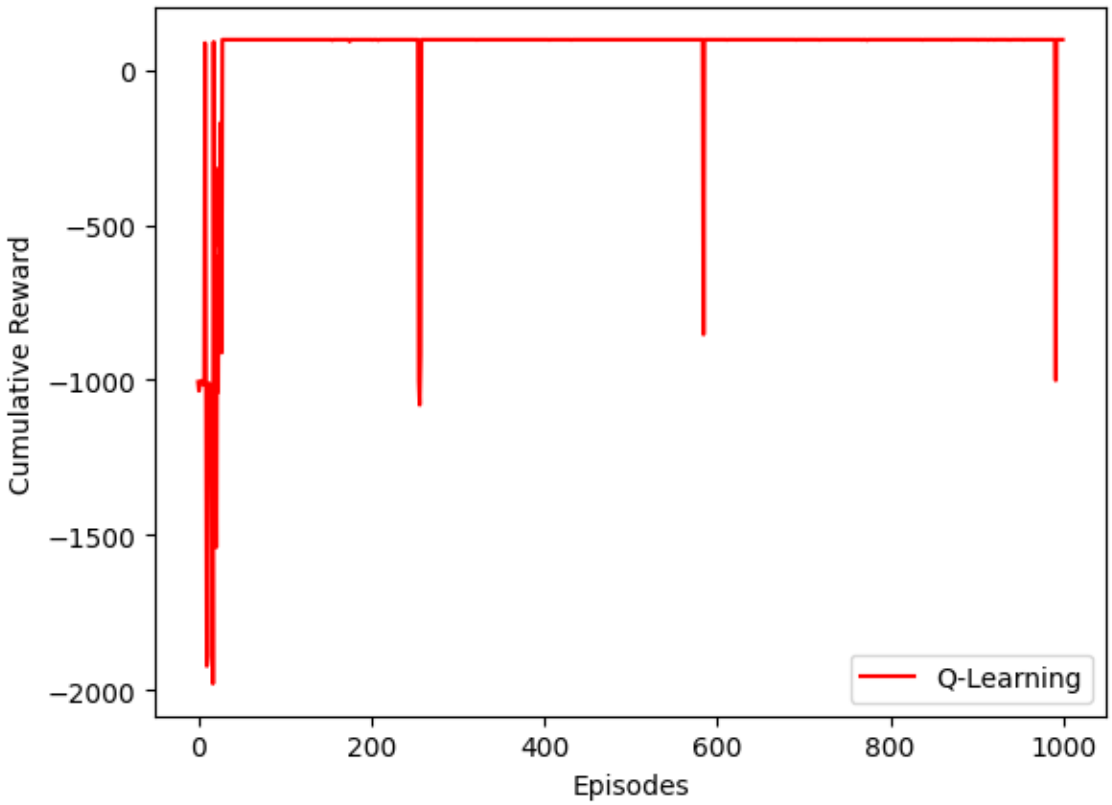
در این قسمت نرخ ϵ را به صورت تطبیقی لحاظ نموده و مشابه بخش قبل دیگر مقدار ثابتی نخواهد داشت که در این حالت الگوریتم Q_Learning توسط کد زیر اصلاح شده است:

```
# Define constants and parameters
grid_size = 4
actions = ['up', 'down', 'left', 'right', 'shoot_up',
'shoot_down', 'shoot_left', 'shoot_right']
alpha = 0.1  # Learning rate
gamma = 0.9  # Discount factor
epsilon = 1.0  # Initial exploration rate
min_epsilon = 0.01  # Minimum exploration rate
epsilon_decay = 0.995  # Exploration rate decay factor

# Initial positions and states
initial_state = (0, 0)
gold_position = (2, 2)  # Example gold position
pit_positions = [(1, 1), (3, 0)]  # Example pit positions
wumpus_position = (3, 3)  # Example Wumpus position
wumpus_alive = True

# Create Q matrix
Q = np.zeros((grid_size, grid_size, len(actions)))

# Function to get next state based on action
def get_next_state(state, action):
    x, y = state
    if action == 'up' and y < grid_size - 1:
        y += 1
    elif action == 'down' and y > 0:
        y -= 1
    elif action == 'left' and x > 0:
        x -= 1
    elif action == 'right' and x < grid_size - 1:
        x += 1
    return (x, y)

# Function to get reward based on next state
def get_reward(state, wumpus_alive):
    if state == gold_position:
        return 100
    elif state in pit_positions:
        return -1000
    elif state == wumpus_position and wumpus_alive:
        return -1000
    else:
        return -1
```

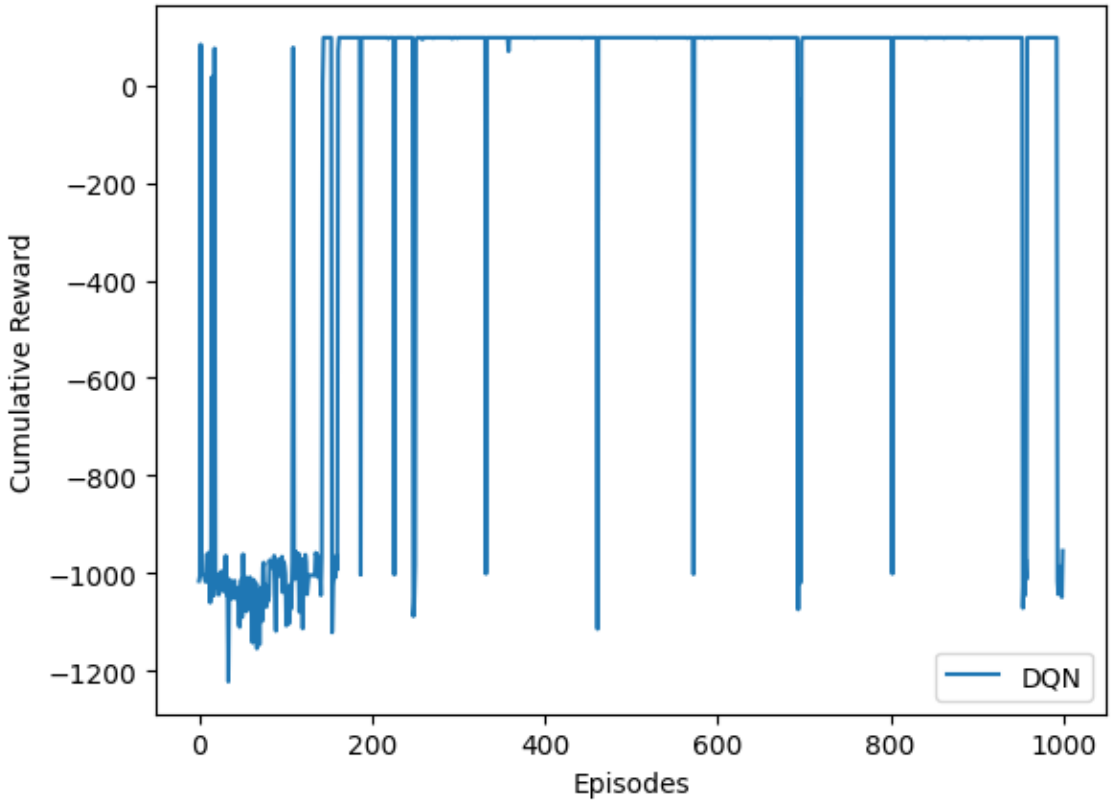
```
# Function to choose action based on epsilon-greedy strategy
def choose_action(state):
    global epsilon
    if np.random.rand() < epsilon:
        return np.random.choice(actions)
    else:
        return actions[np.argmax(Q[state[0], state[1]])]

# Function to update Q values
def update_q(state, action, reward, next_state):
    action_index = actions.index(action)
    best_next_action = np.argmax(Q[next_state[0], next_state[1]])
    td_target = reward + gamma * Q[next_state[0], next_state[1],
best_next_action]
    Q[state[0], state[1], action_index] += alpha * (td_target -
Q[state[0], state[1], action_index])

# Function to simulate Wumpus movement
def move_wumpus(wumpus_position):
    possible_moves = ['up', 'down', 'left', 'right']
    move = random.choice(possible_moves)
    new_position = get_next_state(wumpus_position, move)
    return new_position

# Function to train the agent
def train_agent(episodes):
    global wumpus_alive, wumpus_position, epsilon
    rewards = []
    for _ in range(episodes):
        state = initial_state
        total_reward = 0
        wumpus_alive = True
        wumpus_position = (3, 3)  # Reset Wumpus position
        while True:
            action = choose_action(state)
            next_state = get_next_state(state, action)
            reward = get_reward(next_state, wumpus_alive)

            if 'shoot' in action:
                if next_state == wumpus_position:
                    wumpus_alive = False
                    reward += 50

            update_q(state, action, reward, next_state)
            state = next_state
            total_reward += reward
            if reward == 100 or reward == -1000:
```

```
                break

            # Exploration rate decay
            epsilon = max(min_epsilon, epsilon * epsilon_decay)

            # Move the Wumpus
            wumpus_position = move_wumpus(wumpus_position)

        rewards.append(total_reward)
    return rewards

# Training the Q-learning agent
q_rewards2 = train_agent(1000)
# Show rewards chart
plt.plot(q_rewards2,'r', label='Q-Learning')
plt.xlabel('Episodes')
plt.ylabel('Cumulative Reward')
plt.legend()
plt.show()
```

در این حالت به ازای شرایط قبل نتیجه آموزش به ازای ۱۰۰۰ اپیزود به صورت شکل زیر بدست آمده است:



شکل ۷: پاداش عامل در هر اپیزود با استفاده از روش Q و نرخ تطبیقی ϵ



شکل ۸: پاداش عامل در هر اپیزود با استفاده از روش DeePQ-Learning و نرخ تطبیقی ϵ

در این حالت مطابق شکل های ۷ و ۸ مشاهده می شود هر دو الگوریتم Q و DeepQ توانسته اند عامل را به خوبی آموزش دهند و عامل مسیر بهینه را به خوبی یادگرفته است اما در الگوریتم Q مسیر را بهتر از روش DeepQ یادگرفته است و این امر توسط پاداش های منفی که در شکل ۷ و ۹ که عامل در هر اپیزود دریافت نموده است کاملا مشهود است. مقدار میانگین پاداش ها نیز توسط قطعه کد زیر محاسبه می گردد:

```
q_avg_reward2 = np.mean(q_rewards2)
dqn_avg_reward2 = np.mean(dqn_rewards2)

print(f"Average reward for Q-Learning: {q_avg_reward2}")
print(f"Average reward for DQN: {dqn_avg_reward2}")
```

```
Average reward for Q-Learning: 61.852
Average reward for DQN: -93.429
```

مشاهده می شود عامل در روش الگوریتم Q-Learning پاداش بسیار بهتری از روش DeePQ دریافت نموده است و بیان می کند روش آموزش با الگوریتم Q و نرخ تطبیقی ϵ دارای کارایی بهتری از روش DeePQ بوده است.

د. کارایی یادگیری:

• چند اپیزود طول کشید تا عامل Q-learning به طور مداوم طلا را بدون افتادن در گودال یا خورده شدن توسط Wumpus پیدا کند؟

همانطور که در بالاتر از شـکل ها نتیجه شـد عامل در کمتر از ۵ اپیزود طلا را پیدا می کند اما در اپیزودهای بعدی همچنان توسط Wumpus خورده می شود یا در گودال می افتد که این نشان می دهد عامل بایستی آموزش بیشتری ببیند!

• کارایی یادگیری Q-learning و DQN را مقایسه کنید. کدام یک Policy بهینه را سریعتر یاد گرفت؟

همانطور که بالاتر اشاره کردیم و در شکل ها نیز نشان داده شد؛ الگوریتم DQN نتیجه بهتر و Policy بهتری از الگوریتم Q را نتیجه داد ولی این الگوریتم به دلیل استفاده از ساختار شبکه عصبی بسیار کندتر از الگوریتم Q است و مدت زمان آموزش آن بسیار طولانی تر از الگوریتم Q است لذا در مواردی یا مسائلی که نیاز به سرعت بیشتری هست ممکن است الگوریتم DQN نتواند کارایی خوبی داشته باشد و سراغ الگوریتم Q برویم که سرعت آموزش سریعتری دارد.

ه. معماری شبکه عصبی مورد استفاده برای عامل DQN را شرح دهید. چرا این معماری را انتخاب کردید؟

معماری شبکه عصبی برای عامل DQN

پاسخ:

شبکه عصبی استفاده شده در این کد یک شبکه عصبی چندلایه پرسپترون (MLP) با دو لایه مخفی است. معماری این شبکه به شرح زیر است:

لایه ورودی:

تعداد نورون‌ها: ۲ (این نورون‌ها نمایانگر حالت عامل در محیط هستند، یعنی موقعیت عامل در شبکه)

لایه‌های مخفی:

لایه مخفی اول: ۷۰ نورون، تابع فعال‌سازی tanh

لایه مخفی دوم: ۳۰ نورون، تابع فعال‌سازی tanh

لایه خروجی: خطی

تعداد نورون‌ها: تعداد اکشـــن‌های ممکن (در اینجا ۸ نورون برای ۸ اکشـــن: 'up', 'down', 'left', 'right', 'shoot_up', 'shoot_down', 'shoot_left', 'shoot_right')

چرا این معماری انتخاب شد؟

سادگی و اثربخشی:

شبکه عصبی چندلایه پرسپترون (MLP) با دو لایه مخفی معمولاً برای بسیاری از مسائل مناسب است، زیرا می‌تواند الگوهای پیچیده را بدون پیچیدگی بیش از حد مدل کند.

تعداد نورون‌ها و لایه‌ها به گونه‌ای انتخاب شـــده‌اند که پیچیدگی مدل زیاد نباشـــد و همچنین بتولند روابط غیرخطی بین حالات و اکشن‌ها را یاد بگیرد.

تابع فعال‌سازی tanh:

با توجه به اینکه ورودی ها دارای مقادیر منفی و مثبت بودند ابتدا از تابع فعالســـاز Relu در لایه های پنهان استفاده شد که نتایج رضایت بخش نبودند و این به این دلیل است که این تابع فعالساز بخش منفی را حذف می کند. با تغییر تابع فعالساز به tanh و افزودن بخش منفی در آموزش مدل، شبکه عصبی به خوبی آموزش دید و نتایج بهتری بدست آمد.

شایان ذکر است که کدهای یکپارچه این بخش در این لینک قرار دارد

منابع:

1. https://github.com/ArmanMarzban/MJAHMADEE